Computational Thinking

http://www.bbc.co.uk/education/guides/zp92mp3/revision/1

Introduction to computational thinking

Before computers can be used to solve a problem, the problem itself and the ways in which it could be resolved must be understood. Computational thinking techniques help with these tasks.

What is computational thinking?

Computers can be used to help us solve problems. However, before a problem can be tackled, the problem itself and the ways in which it could be solved need to be understood.

Computational thinking allows us to do this.

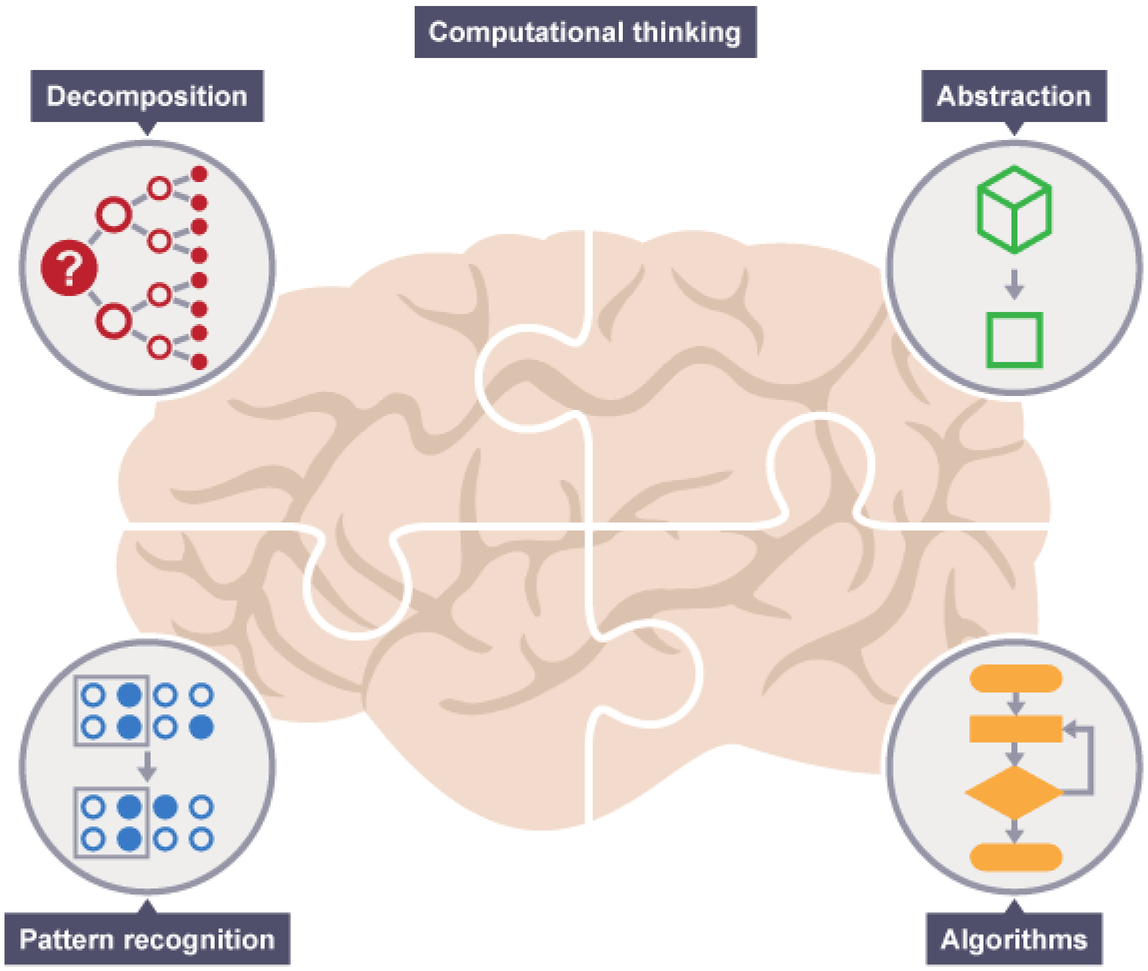
Computational thinking allows us to take a complex problem, understand what the problem is and develop possible solutions. We can then present these solutions in a way that a computer, a human, or both, can understand.

The four cornerstones of computational thinking

There are four key techniques (cornerstones) to computational thinking:

- **decomposition** - breaking down a complex problem or system into smaller, more manageable parts
- **pattern recognition** – looking for similarities among and within problems
- **abstraction** – focusing on the important information only, ignoring irrelevant detail
- **algorithms** - developing a step-by-step solution to the problem, or the rules to follow to solve the problem

Each cornerstone is as important as the others. They are like legs on a table - if one leg is missing, the table will probably collapse. Correctly applying all four techniques will help when programming a computer.



Computational thinking in practice

A complex problem is one that, at first glance, we don't know how to solve easily.
Computational thinking involves taking that complex problem and breaking it down into a series of small, more manageable problems (**decomposition**). Each of these smaller problems can then be looked at individually, considering how similar problems have been solved previously (**pattern recognition**) and focusing only on the important details, while ignoring irrelevant information (**abstraction**). Next, simple steps or rules to solve each of the smaller problems can be designed (**algorithms**).
Finally, these simple steps or rules are used to **program** a computer to help solve the complex problem in the best way.

http://www.bbc.co.uk/education/guides/zp92mp3/revision/2

Thinking computationally

Thinking computationally is not **programming**. It is not even thinking like a computer, as computers do not, and cannot, think.

Simply put, programming tells a computer what to do and how to do it.**Computational thinking enables you to work out exactly what to tell the computer to do.**
For example, if you agree to meet your friends somewhere you have never been before, you would probably plan your route before you step out of your house. You might consider the routes available and which route is ‘best’ - this might be the route that is the shortest, the quickest, or the one which goes past your favourite shop on the way. You'd then follow the step-by-step directions to get there. In this case,**the planning part is like computational thinking,** and **following the directions is like programming.**

Being able to turn a complex problem into one we can easily understand is a skill that is extremely useful. In fact, it's a skill you already have and probably use every day.

For example, it might be that you need to decide what to do with your group of friends. If all of you like different things, you would need to decide:

- what you could do
- where you could go
- who wants to do what
- what you have previously done that has been a success in the past
- how much money you have and the cost of any of the options
- what the weather might be doing
- how much time you have

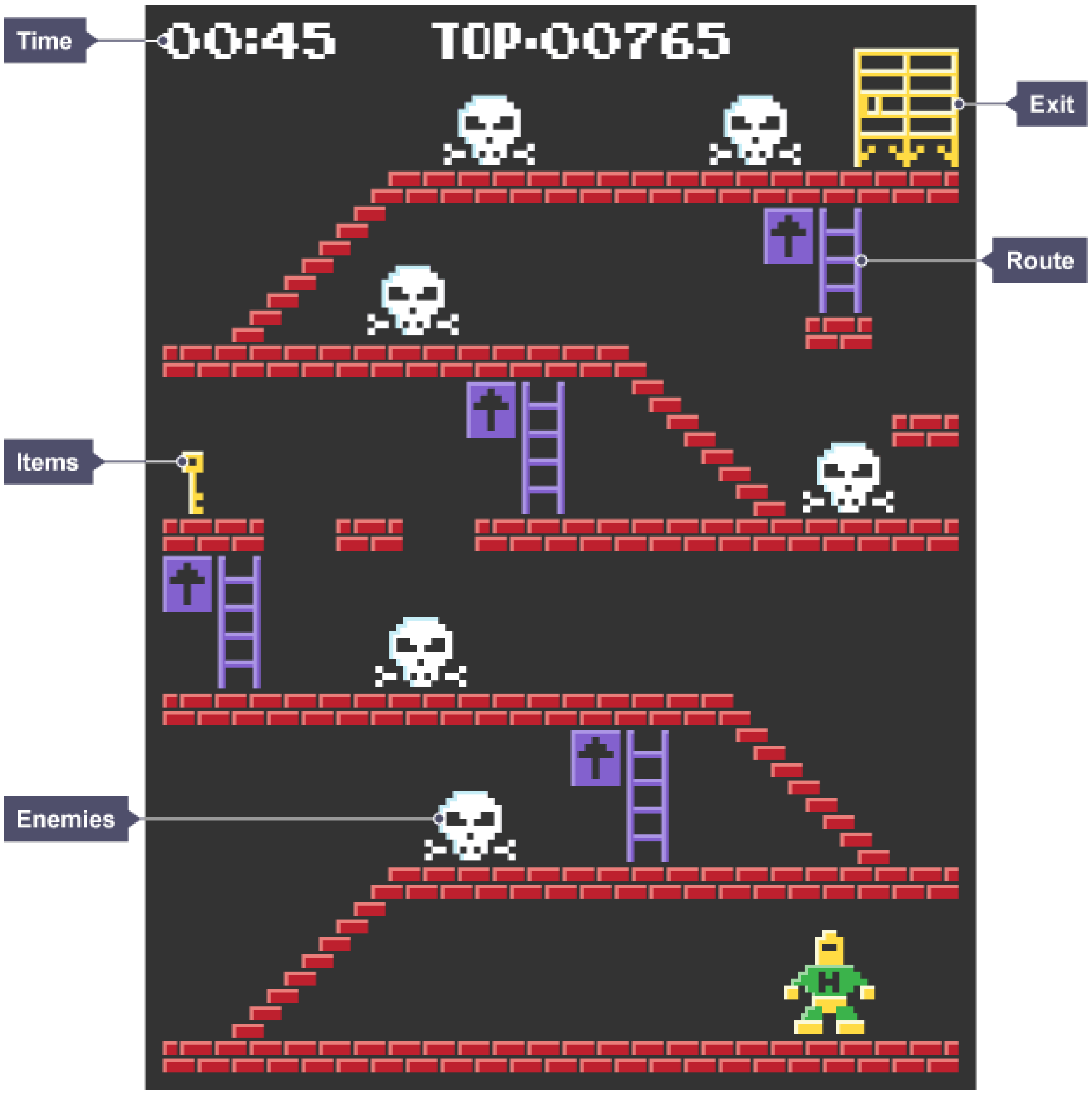
From this information, you and your friends could decide more easily where to go and what to do – in order to keep most of your friends happy. You could also use a computer to help you to collect and analyse the data to devise the best solution to the problem, both now and if it arose again in the future, if you wished.

Another example might occur when playing a videogame. Depending on the game, in order to complete a level you would need to know:

- what items you need to collect, how you can collect them, and how long you have in which to collect them
- where the exit is and the best route to reach it in the quickest time possible
- what kinds of enemies there are and their weak points

From these details you can work out a strategy for completing the level in the most efficient way.

If you were to create your own computer game, these are exactly the types of questions you would need to think about and answer before you were able to program your game.



Both of the above are examples of where computational thinking has been used to solve a complex problem:

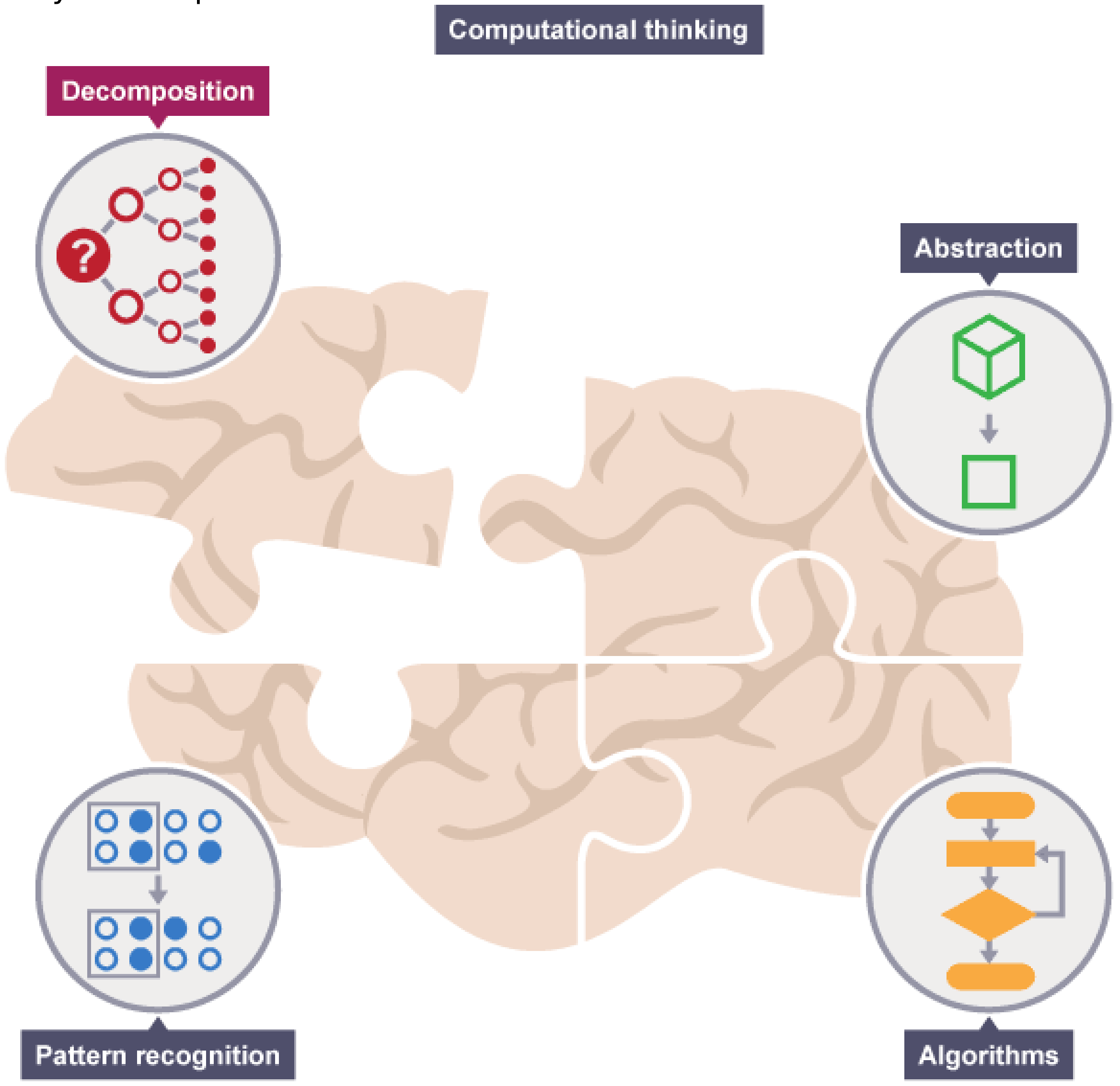
- each complex problem was broken down into several small decisions and steps (eg where to go, how to complete the level – **decomposition**)
- only the relevant details were focused on (eg weather, location of exit – **abstraction**)
- knowledge of previous similar problems was used (**pattern recognition**...
- ...to work out a step by step plan of action (**algorithms**)

Decomposition

Before computers can solve a problem, the problem and the ways in which it can be resolved must be understood. Decomposition helps by breaking down complex problems into more manageable parts.

What is decomposition?

Decomposition is one of the four cornerstones of Computer Science. It**involves breaking down a complex problem or system into smaller parts that are more manageable and easier to understand**. The smaller parts can then be examined and solved, or designed individually, as they are simpler to work with.



Why is decomposition important?

If a problem is not decomposed, it is much harder to solve. Dealing with many different stages all at once is much more difficult than breaking a problem down into a number of smaller problems and solving each one, one at a time. Breaking the problem down into smaller parts means that each smaller problem can be examined in more detail.

Similarly, trying to understand how a complex system works is easier using decomposition. For example, understanding how a bicycle works is more straightforward if the whole bike is separated into smaller parts and each part is examined to see how it works in more detail.

Decomposition in practice

We do many tasks on a daily basis without even thinking about – or decomposing – them, such as brushing our teeth.

Example 1: Brushing our teeth

To decompose the problem of how to brush our teeth, we would need to consider:

- which toothbrush to use
- how long to brush for
- how hard to press on our teeth
- what toothpaste to use

Example 2: Solving a crime

It is only normally when we are asked to do a new or more complex task that we start to think about it in detail – to **decompose** the task.

Imagine that a crime has been committed. Solving a crime can be a very complex problem as there are many things to consider.

For example, a police officer would need to know the answer to a series of smaller problems:

- what crime was committed
- when the crime was committed
- where the crime was committed
- what evidence there is
- if there were any witnesses
- if there have recently been any similar crimes

The complex problem of the committed crime has now been broken down into simpler problems that can be examined individually, in detail.

Decomposing creating an app

Imagine that you want to create your first app. This is a complex problem - there are lots of things to consider.

Question

How would you decompose the task of creating an app?

Reveal answer

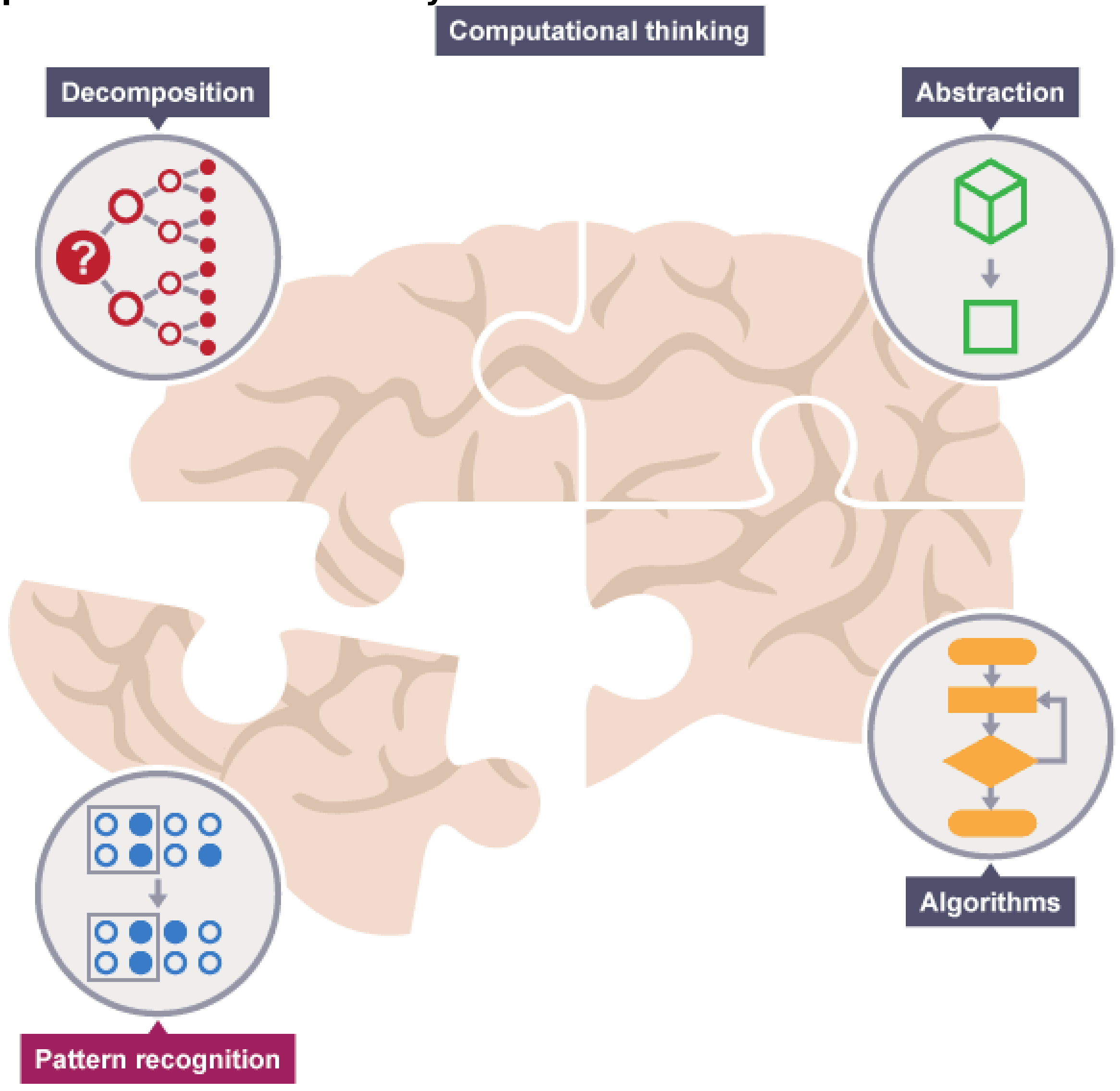
To **decompose** this task, you would need to know the answer to a series of smaller problems:

- what kind of app you want to create
- what your app will look like
- who the target audience for your app is
- what your graphics will look like
- what audio you will include
- what software you will use to build your app
- how the user will navigate your app
- how you will test your app
- where you will sell your app

This list has broken down the complex problem of creating an app into much simpler problems that can now be worked out. You may also be able to get other people to help you with different individual parts of the app. For example, you may have a friend who can create the graphics, while another will be your tester.

What is pattern recognition?

When we decompose a complex problem we often find patterns among the smaller problems we create. The patterns are similarities or characteristics that some of the problems share.
Pattern recognition is one of the four cornerstones of Computer Science. It **involves finding the similarities or patterns among small, decomposed problems that can help us solve more complex problems more efficiently**.

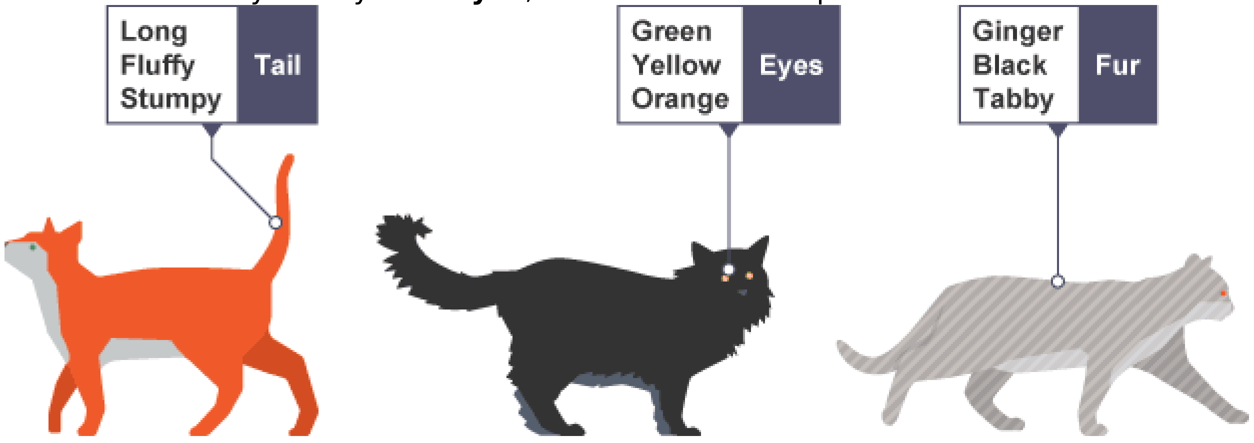


What are patterns?

Imagine that we want to draw a series of cats.
All cats share common characteristics. Among other things **they all have eyes, tails and fur**. They also like to eat fish and make meowing sounds.

Because we know that all cats have eyes, tails and fur, we can make a good attempt at drawing a cat, simply by including these common characteristics. In computational thinking, these characteristics are known as patterns. **Once we know how to describe one cat we can describe others, simply by following this pattern.** The only things that are different are the specifics:

- one cat may have green **eyes**, a long **tail** and black **fur**
- another cat may have yellow **eyes**, a short **tail** and striped **fur**



<

>

Why do we need to look for patterns?

Finding patterns is extremely important. Patterns make our task simpler. Problems are easier to solve when they share patterns, because we can use the same problem-solving solution wherever the pattern exists.

The more patterns we can find, the easier and quicker our overall task of problem solving will be.

If we want to draw a number of cats, finding a pattern to describe cats in general, eg they all have eyes, tails and fur, makes this task quicker and easier.

We know that all cats follow this pattern, so we don't have to stop each time we start to draw a new cat to work this out. From the patterns we know cats follow, we can quickly draw several cats.

What happens when we don’t look for patterns?

Suppose we hadn’t looked for patterns in cats. Each time we wanted to draw a cat, we would have to stop and work out what a cat looked like. This would slow us down.

We could still draw our cats - and they would look like cats - but each cat would take far longer to draw. This would be very inefficient, and a poor way to go about solving the cat-drawing task.

In addition, if we don’t look for patterns we might not realise that all cats have eyes, tails and fur. When drawn, our cats might not even look like cats. In this case, because we didn’t recognise the pattern, we would be solving the problem incorrectly.

Recognising patterns

To find patterns in problems we look for things that are the same (or very similar) in each problem. It may turn out that no common characteristics exist among problems, but we should still look.
Patterns exist **among different problems** and **within individual problems**. We need to look for both.

Patterns among different problems

To find patterns among problems we look for things that are the same (or very similar) for each problem.
For example, **decomposing** the task of baking a cake would highlight the need for us to know the solutions to a series of smaller problems:

- what kind of cake we want to bake
- what ingredients we need and how much of each

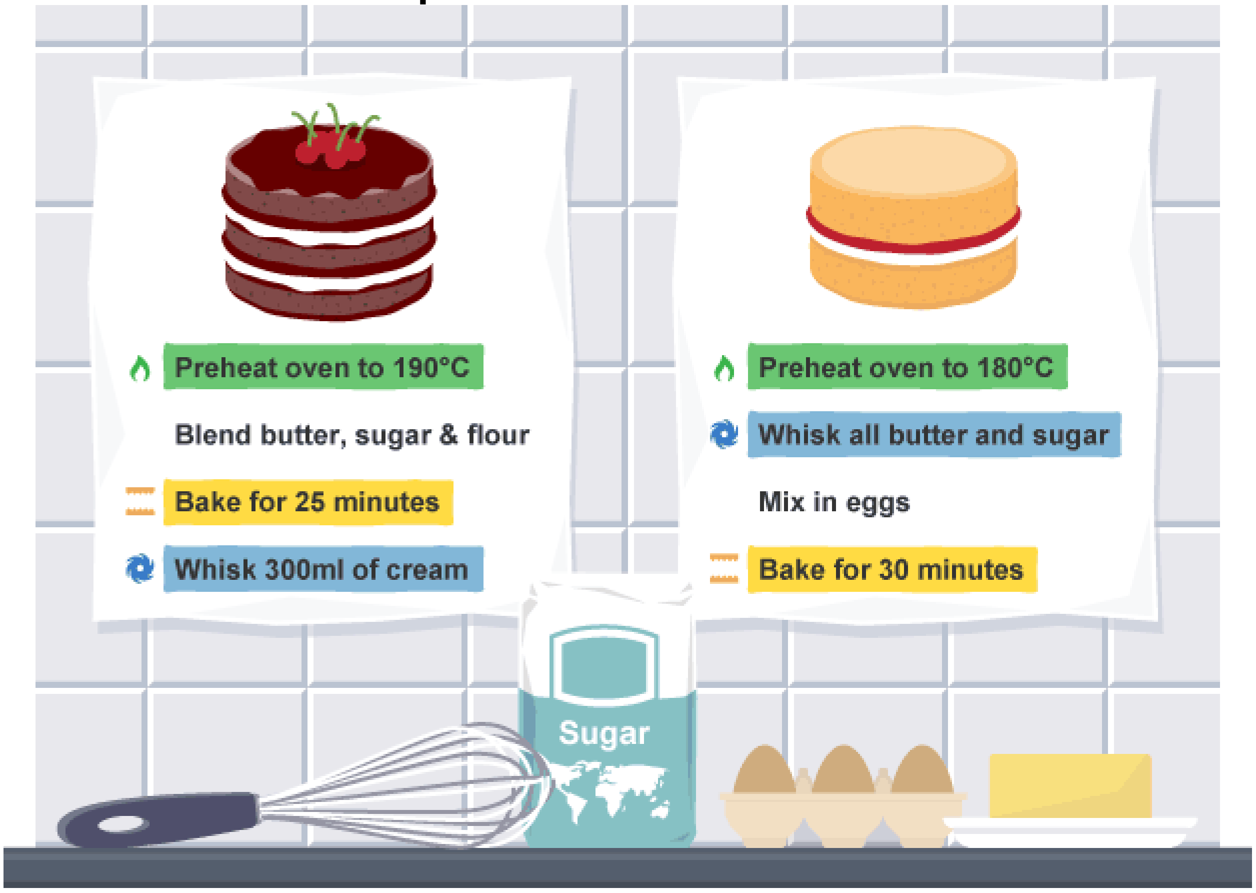
- how many people we want to bake the cake for
- how long we need to bake the cake for
- when we need to add each ingredient
- what equipment we need

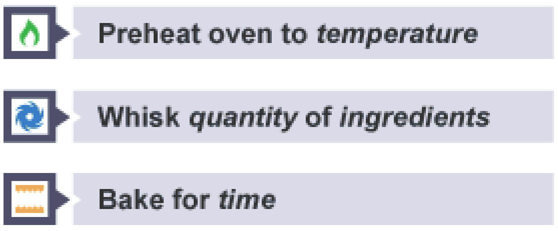
Once we know how to bake one particular type of cake, we can see that baking another type of cake is not that different - because patterns exist.

For example:

- each cake will need a precise quantity of specific ingredients
- ingredients will get added at a specific time
- each cake will bake for a specific period of time

Once we have the patterns identified, we can work on common solutions between the problems.




Preheat oven to *temperature*
Whisk *quantity of ingredients*
Bake for *time*

Patterns within problems

Patterns may also exist within the smaller problems we have decomposed to.

If we look at baking a cake, we can find patterns within the smaller problems, too. For example, for ‘each cake will need a precise quantity of specific ingredients’, each ingredient needs:

- identifying (naming)
- a specific measurement

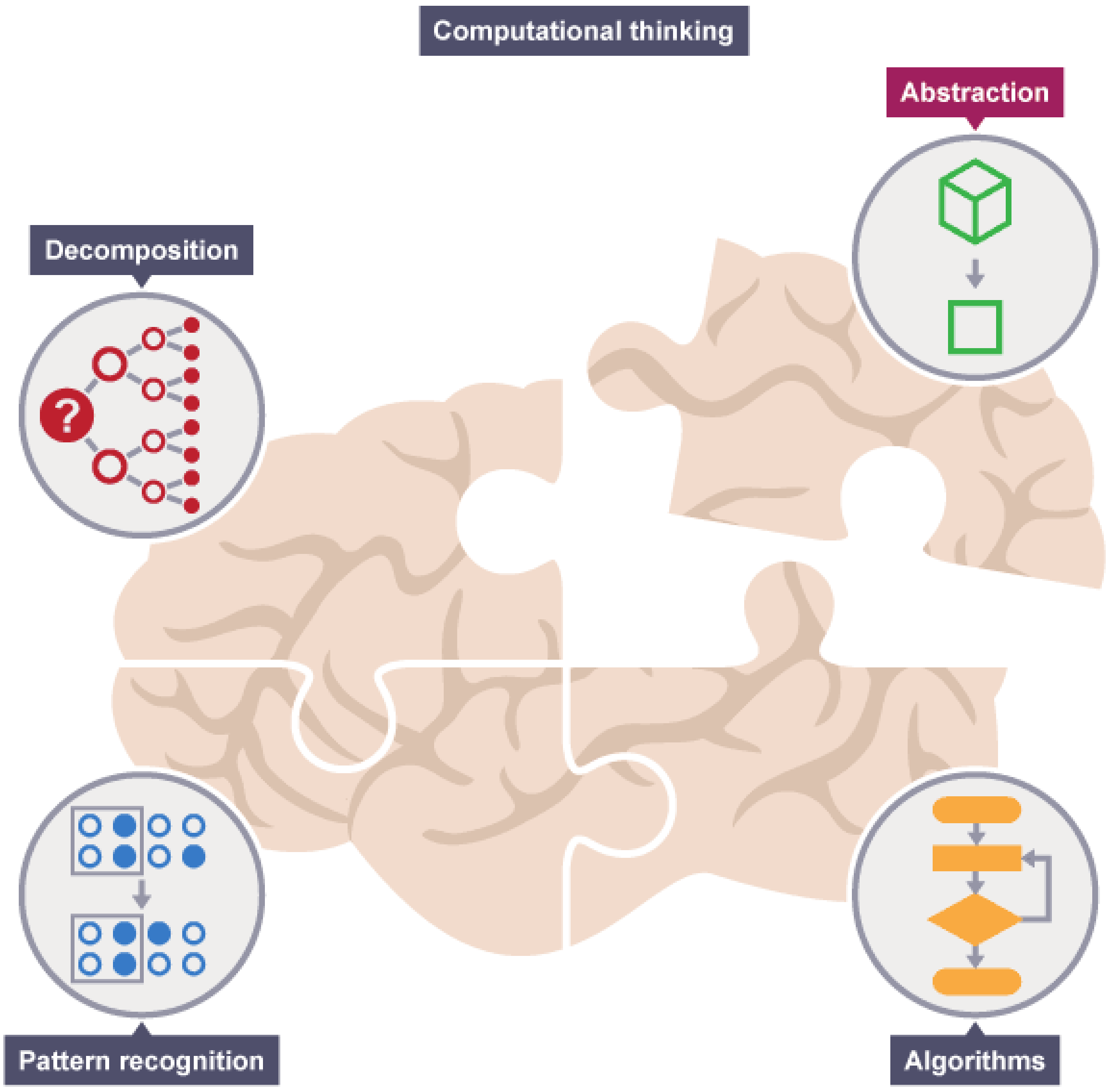
Once we know how to identify each ingredient and its amount, we can apply that pattern to all ingredients. Again, all that changes is the specifics.

Abstraction

Once we have recognised patterns in our problems, we use abstraction to gather the general characteristics and to filter out of the details we do not need in order to solve our problem.

What is abstraction?

Abstraction is one of the four cornerstones of Computer Science. It involves filtering out – essentially, ignoring - the characteristics that we don't need in order to concentrate on those that we do.



In **computational thinking**, when we **decompose** problems, we then look for patterns among and within the smaller problems that make up the complex problem.

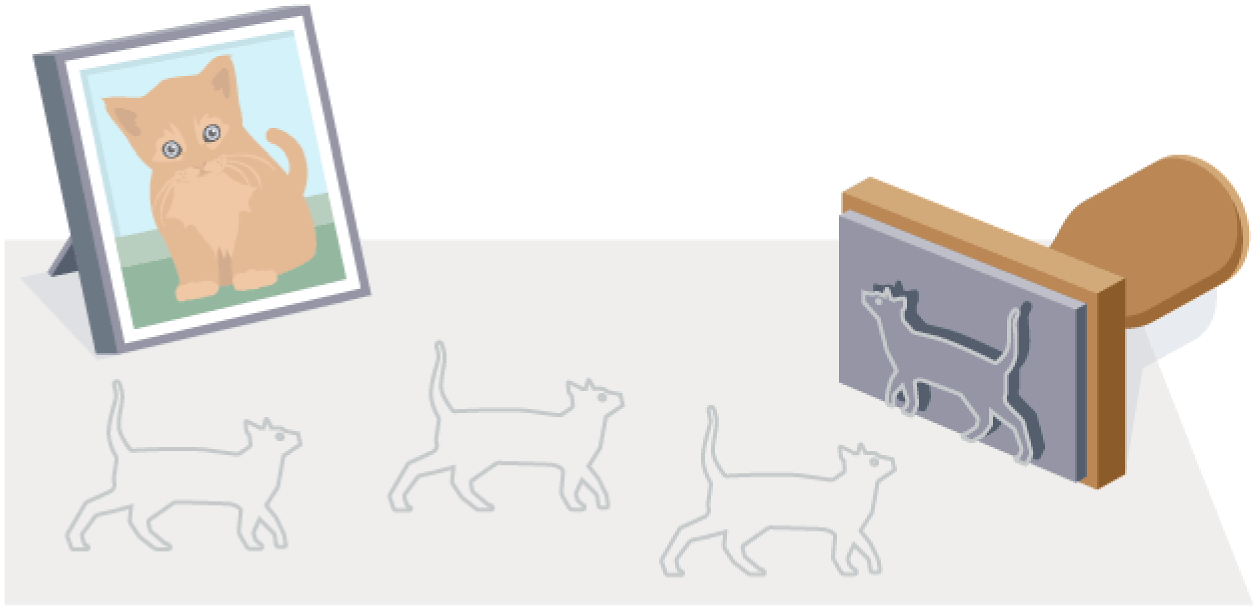
Abstraction is the process of filtering out – ignoring - the characteristics of patterns that we don't need in order to concentrate on those that we do. It is also the filtering out of specific details. From this we create a representation (idea) of what we are trying to solve.

What are specific details or characteristics?

In **pattern recognition** we looked at the problem of having to draw a series of cats.
We noted that all cats have general characteristics, which are common to all cats, eg eyes, a tail, fur, a liking for fish and the ability to make meowing sounds. In addition, each cat has **specific characteristics**, such as **black** fur, a **long** tail, **green** eyes, a love of **salmon**, and a**loud** meow. **These details are known as specifics**.

In order to draw a basic cat, we **do** need to know that it has a tail, fur and eyes. These characteristics are relevant. We **don't** need to know what sound a cat makes or that it likes fish. These characteristics are irrelevant and can be filtered out. We **do** need to know that a cat has a tail, fur and eyes, but we **don't** need to know what size and colour these are. These specifics can be filtered out.

From the general characteristics we have (tail, fur, eyes) we can build a basic idea of a cat, ie what a cat basically looks like. Once we know what a cat looks like we can describe how to draw a basic cat.



Why is abstraction important?

Abstraction allows us to create a general idea of what the problem is and how to solve it. The process instructs us to remove all specific detail, and any patterns that will not help us solve our problem. This helps us form our idea of the problem. This idea is known as a ‘model’.

If we don’t abstract we may end up with the wrong solution to the problem we are trying to solve. With our cat example, if we didn’t abstract we might think that all cats have long tails and short fur. Having abstracted, we know that although cats have tails and fur, not all tails are long and not all fur is short. In this case, abstraction has helped us to form a clearer model of a cat.

<

How to abstract

Abstraction is the gathering of the general characteristics we need and the filtering out of the details and characteristics that we do not need.

When baking a cake, there are some general characteristics between cakes. For example:

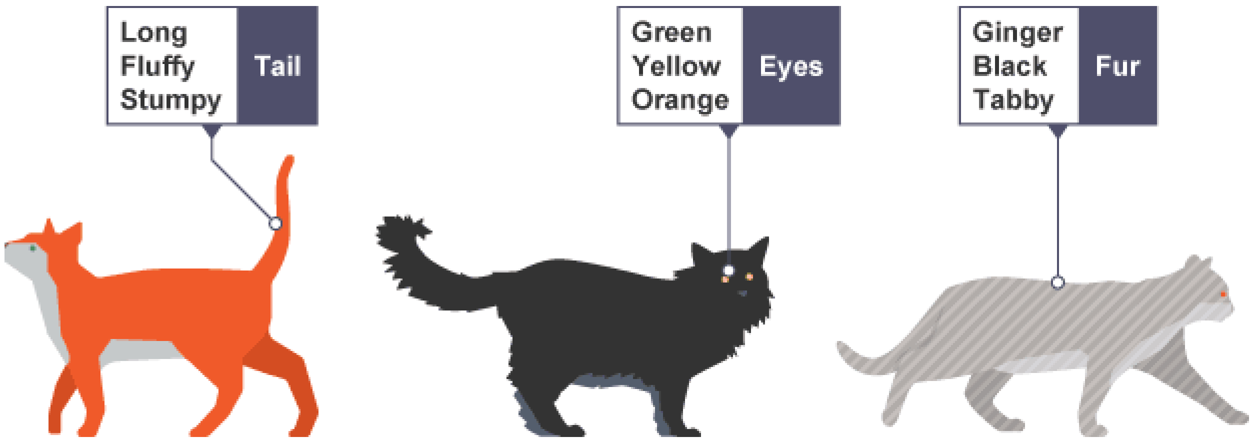
- a cake needs ingredients
- each ingredient needs a specified quantity
- a cake needs timings

When abstracting, we remove specific details and keep the general relevant patterns.

General patterns	Specific details
We need to know that a cake has ingredients	We don't need to know what those ingredients are
We need to know that each ingredient has a specified quantity	We don’t need to know what that quantity is
We need to know that each cake needs a specified time to bake	We don't need to know how long the time is

Creating a model

A model is a general idea of the problem we are trying to solve.
For example, a model cat would be any cat. Not a specific cat with a long tail and short fur - **the model represents all cats**. From our model of cats, we can learn what any cat looks like, using the patterns all cats share.



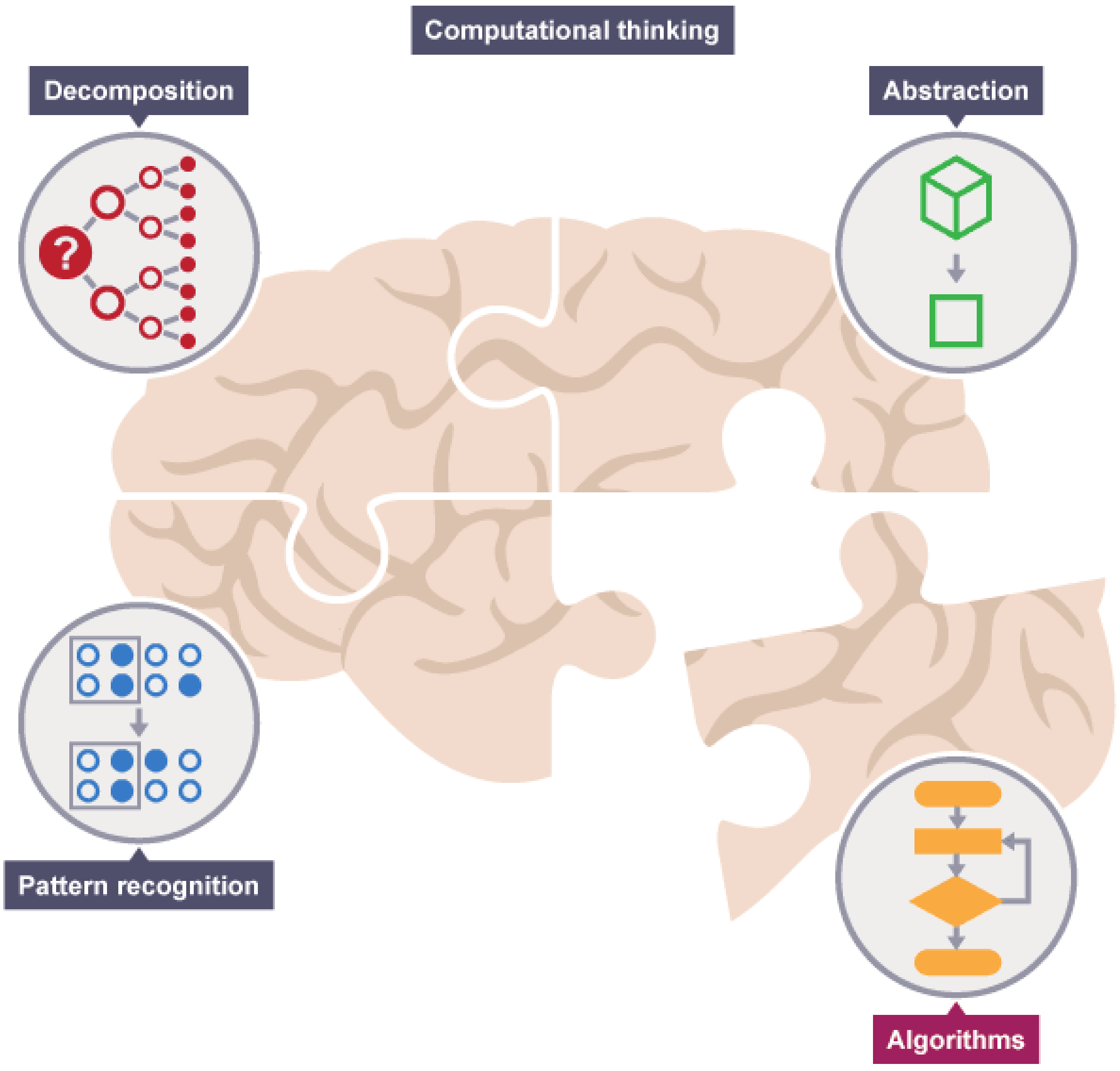
Similarly, when baking a cake, a model cake wouldn't be a specific cake, like a sponge cake or a fruit cake. Instead, the model would represent all cakes. From this model we can learn how to bake any cake, using the patterns that apply to all cakes.
Once we have a model of our problem, we can then design an **algorithm** to solve it.

Algorithms

An algorithm is a plan, a set of step-by-step instructions to resolve a problem. In an algorithm, each instruction is identified and the order in which they should be carried out is planned.

What is an algorithm?

Algorithms are one of the four cornerstones of Computer Science. **An algorithm is a plan, a set of step-by-step instructions to solve a problem.** If you can tie shoelaces, make a cup of tea, get dressed or prepare a meal then you already know how to follow an algorithm.



In an algorithm, each **instruction** is identified and the order in which they should be carried out is planned. Algorithms are often used as a starting point for creating a computer program, and they are sometimes written as a **flowchart** or in **pseudocode**.
If we want to tell a computer to do something, we have to write a computer program that will tell the computer, step-by-step, exactly what we want it to do and how we want it to do it. **This step-by-step program will need planning, and to do this we use an algorithm.**

Computers are only as good as the algorithms they are given. If you give a computer a poor algorithm, you will get a poor result – hence the phrase: ‘Garbage in, garbage out.’

Algorithms are used for many different things including calculations, data processing and automation.

Making a plan

It is important to plan out the solution to a problem to make sure that it will be correct. Using **computational thinking** and **decomposition**we can break down the problem into smaller parts and then we can plan out how they fit back together in a suitable order to solve the problem.

This order can be represented as an algorithm. An algorithm must be clear. It must have a starting point, a finishing point and a set of clear instructions in between.

Representing an algorithm: Pseudocode

There are two main ways that **algorithms** can be represented – **pseudocode** and **flowcharts**.
Most **programs** are developed using **programming languages**. These languages have specific **syntax** that must be used so that the program will run properly. **Pseudocode is not a programming language**, it is a simple way of describing a set of instructions that does not have to use specific syntax.
Writing in pseudocode is similar to writing in a programming language. Each step of the algorithm is written on a line of its own in sequence.**Usually, instructions are written in uppercase**, **variables in lowercase and messages in sentence case**.
In pseudocode, **INPUT** asks a question. **OUTPUT** prints a message on screen.

A simple program could be created to ask someone their name and age, and to make a comment based on these. This program represented in pseudocode would look like this:

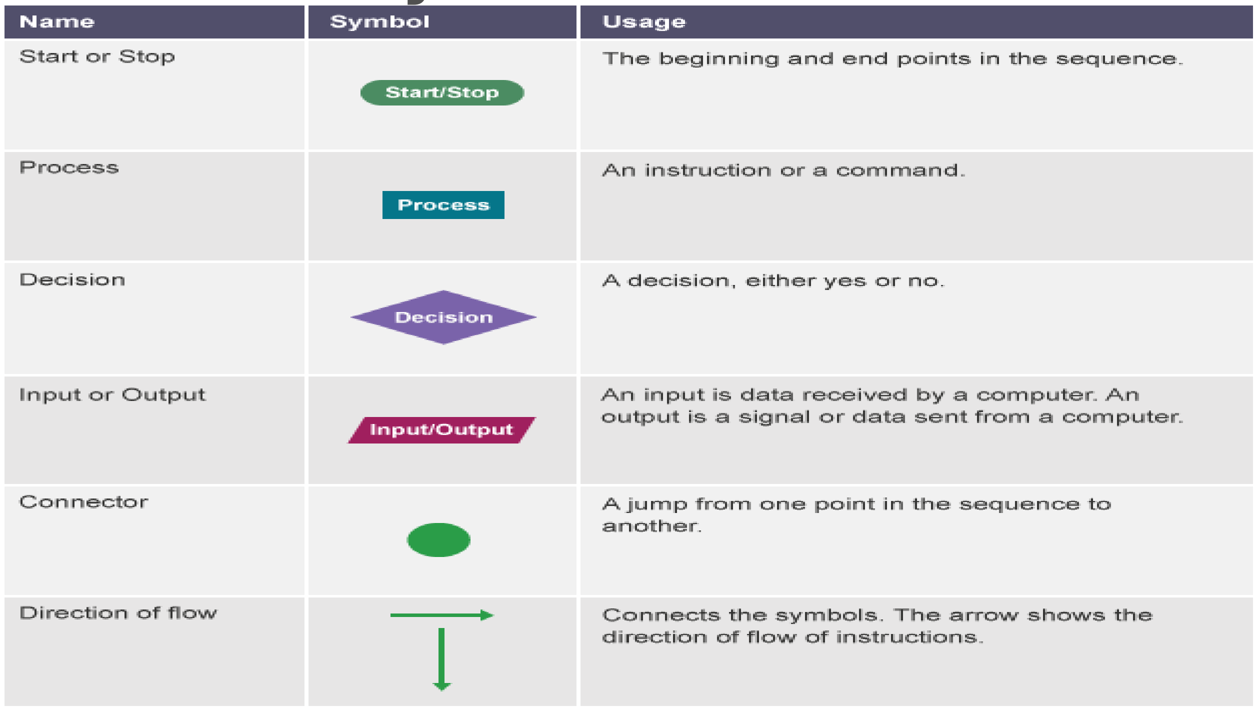
```
OUTPUT 'What is your name?'
INPUT user inputs their name
STORE the user's input in the name variable
OUTPUT 'Hello' + name
OUTPUT 'How old are you?'
INPUT user inputs their age
STORE the user's input in the age variable
IF age >= 70 THEN
       OUTPUT 'You are aged to perfection!'
ELSE
       OUTPUT 'You are a spring chicken!'
```

In programming, > means ‘greater than’, < means ‘less than’, ≥ means ‘greater than or equal to’ and ≤ means ‘less than or equal to’.

Representing an algorithm: Flowcharts

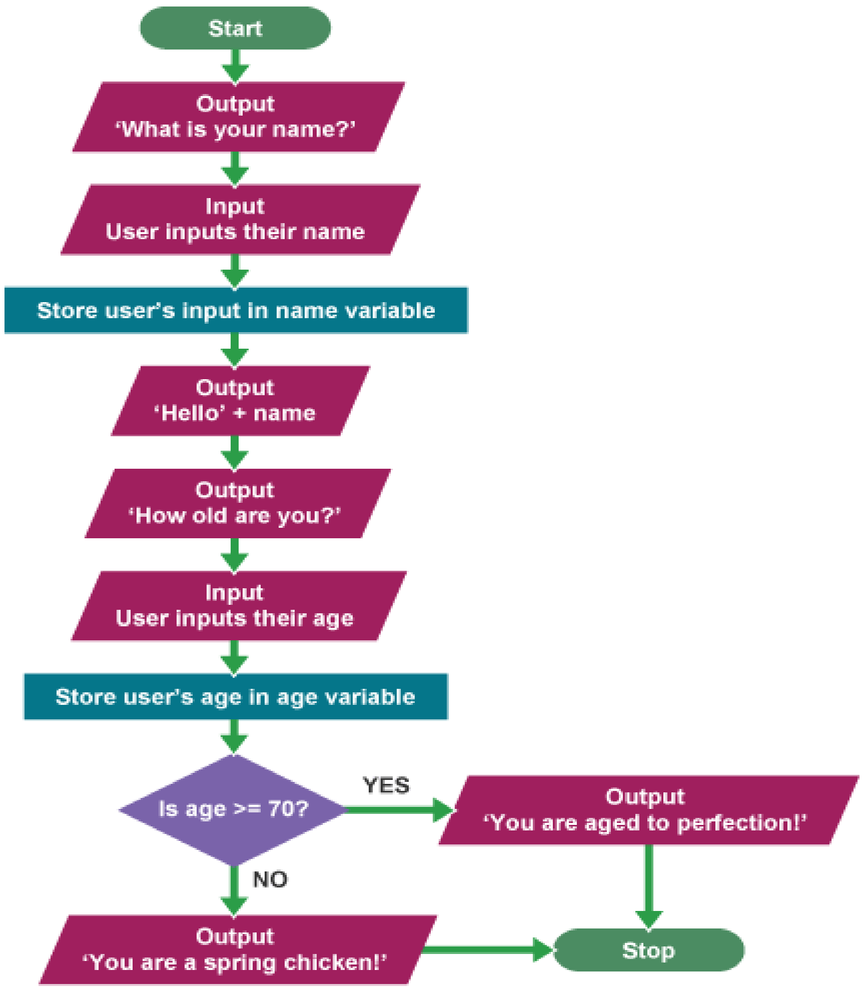
A flowchart is a diagram that represents a set of **instructions**. Flowcharts normally use standard symbols to represent the different instructions. There are few real rules about the level of detail needed in a flowchart. Sometimes flowcharts are broken down into many steps to provide a lot of detail about exactly what is happening. Sometimes they are simplified so that a number of steps occur in just one step.

Flowchart symbols



Name	Symbol	Usage
Start or Stop	Start/Stop	The beginning and end points in the sequence.
Process	Process	An instruction or a command.
Decision	Decision	A decision, either yes or no.
Input or Output	Input/Output	An input is data received by a computer. An output is a signal or data sent from a computer.
Connector	●	A jump from one point in the sequence to another.
Direction of flow	→ ↓	Connects the symbols. The arrow shows the direction of flow of instructions.

A simple **program** could be created to ask someone their name and age, and to make a comment based on these. This program represented as a flowchart would look like this:



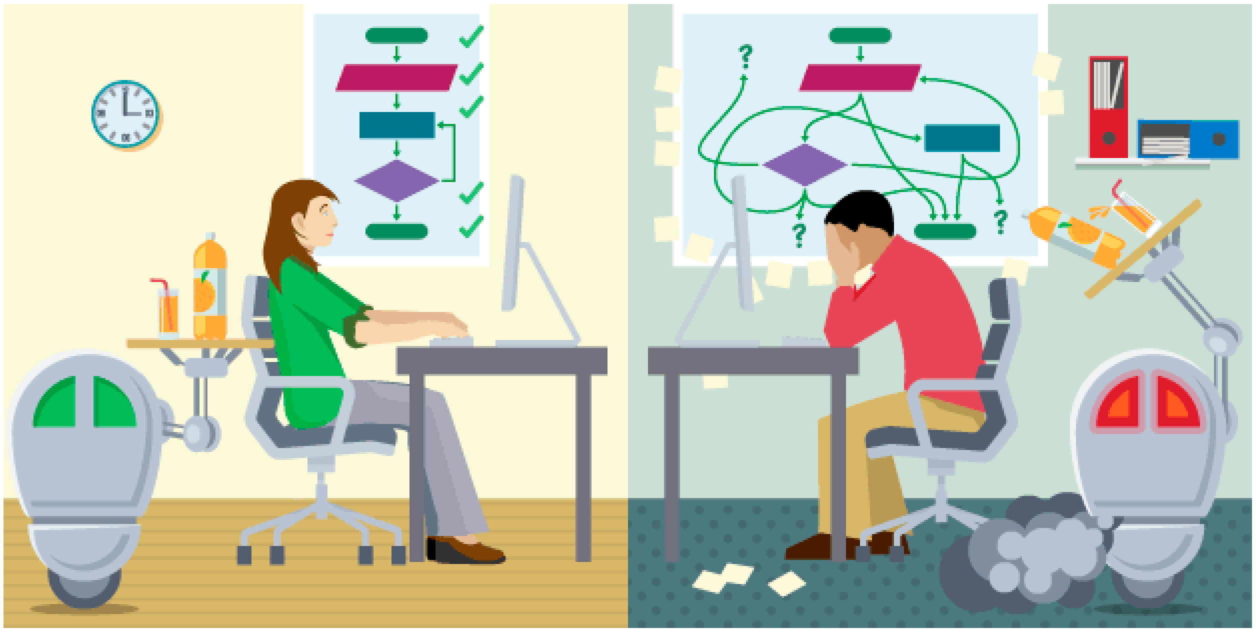
What is evaluation?

Once a solution has been designed using **computational thinking**, it is important to make sure that the solution is fit for purpose.

Evaluation is the process that allows us to make sure our solution does the job it has been designed to do and to think about how it could be improved. Once written, an **algorithm** should be checked to make sure it:

- is easily understood – is it fully **decomposed**?
- is complete – does it solve every aspect of the problem?
- is efficient – does it solve the problem, making best use of the available resources (eg as quickly as possible/using least space)?
- meets any **design criteria** we have been given

If an algorithm meets these four criteria it is likely to work well. The algorithm can then be programmed.
Failure to evaluate can make it difficult to write a program. Evaluation helps to make sure that as few difficulties as possible are faced when programming the solution.



Why do we need to evaluate our solutions?

Computational thinking helps to solve problems and design a solution – an **algorithm** – that can be used to program a computer. However, if the solution is faulty, it may be difficult to write the program. Even worse, the finished program might not solve the problem correctly.

Evaluation allows us to consider the solution to a problem, make sure that it meets the original design criteria, produces the correct solution and is fit for purpose - before programming begins.

What happens if we don’t evaluate our solutions?

Once a solution has been decided and the algorithm designed, it can be tempting to miss out the evaluating stage and to start**programming** immediately. **However, without evaluation any faults in the algorithm will not be picked up, and the program may not correctly solve the problem, or may not solve it in the best way.**

Faults may be minor and not very important. For example, if a solution to the question ‘how to draw a cat?’ was created and this had faults, all that would be wrong is that the cat drawn might not look like a cat. However, faults can have huge – and terrible – effects, eg if the solution for an aeroplane autopilot had faults.

Ways that solutions can be faulty

We may find that solutions fail because:

- it is **not fully understood** - we may not have properly **decomposed** the problem
- it is **incomplete** - some parts of the problem may have been left out accidentally
- it is **inefficient** – it may be too complicated or too long
- it **does not meet the original design criteria** – so it is not fit for purpose

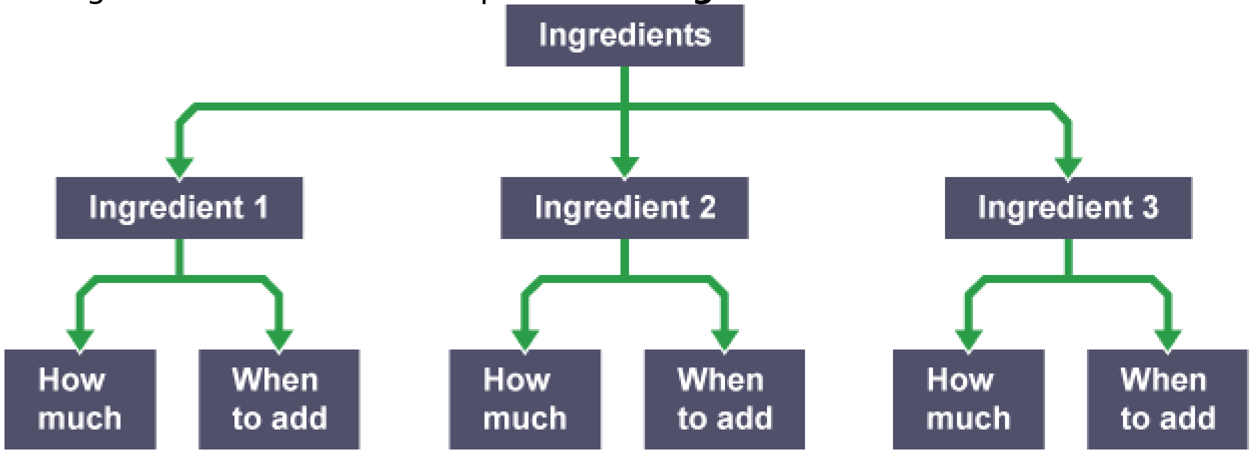
A faulty solution may include one or more of these errors.

Solutions that are not properly decomposed

If **computational thinking** techniques are applied to the problem of how to bake a cake, on **decomposing** the problem, it is necessary to know:

- what kind of cake to bake
- what ingredients are needed, how much of each ingredient, and when to add it
- how many people the cake is for
- how long to bake the cake for
- what equipment is needed

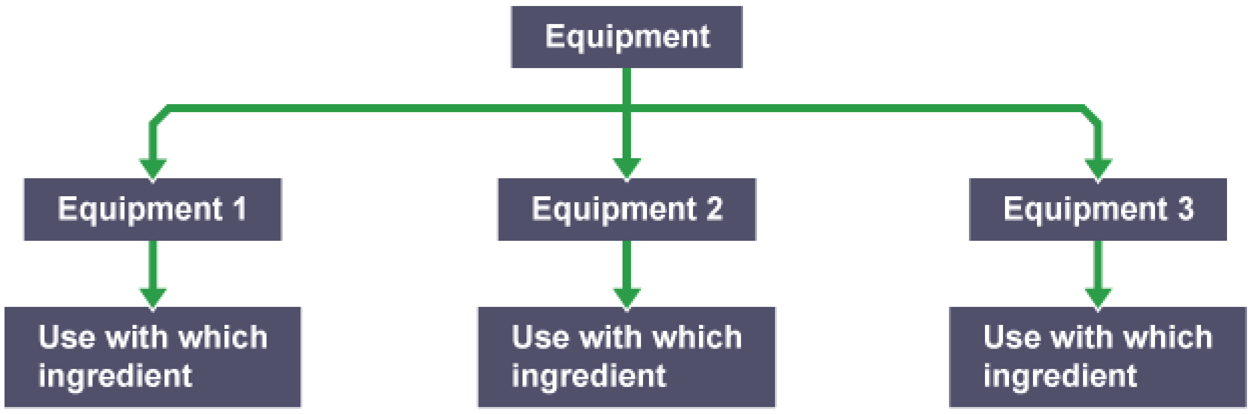
A diagram of a further decomposition of **ingredients** would look like this:



At the moment, a diagram of the further decomposition of **equipment**would look like this:

The ‘Equipment’ part is not properly broken down (or decomposed). Therefore, if the solution - or **algorithm** – were created from this, baking the cake would run into problems. The algorithm would say what equipment is needed, but not how to use it, so a person could end up trying to use a knife to measure out the flour and a whisk to cut a lump of butter, for example. This would be wrong and would, of course, not work.

Ideally, then, ‘Equipment’ should be decomposed further, to state which equipment is needed and which ingredients each item is used with.


Equipment
Equipment 1
Equipment 2
Equipment 3
Use with which ingredient
Use with which ingredient
Use with which ingredient

The problem occurred here because the problem of which equipment to use and which ingredients to use it with hadn't been fully decomposed.

Solutions that are incomplete

If **computational thinking** techniques are applied to the problem of how to bake a cake, on **decomposing** the problem, it is necessary to know:

- what kind of cake to bake
- what ingredients are needed, how much of each ingredient, and when to add it
- how many people the cake is for
- how long to bake the cake for
- what equipment is needed

However, **this is incomplete** – part of the problem has been left out. We still need to know:

- where to bake the cake
- what temperature to bake the cake at

Therefore, if this information was used to create the solution, the**algorithm** would say how long the cake should be baked for but it would not state that the cake should be placed in the oven, or the temperature that the oven should be. Even if the cake made it to the oven, it could end up undercooked or burnt to a cinder.

Very important factors have been left out, so the chances of making a great cake are slim.

The problem occurred here because placing the cake in the oven and specifying the oven temperature had not been included, making the solution incomplete.

&lt;

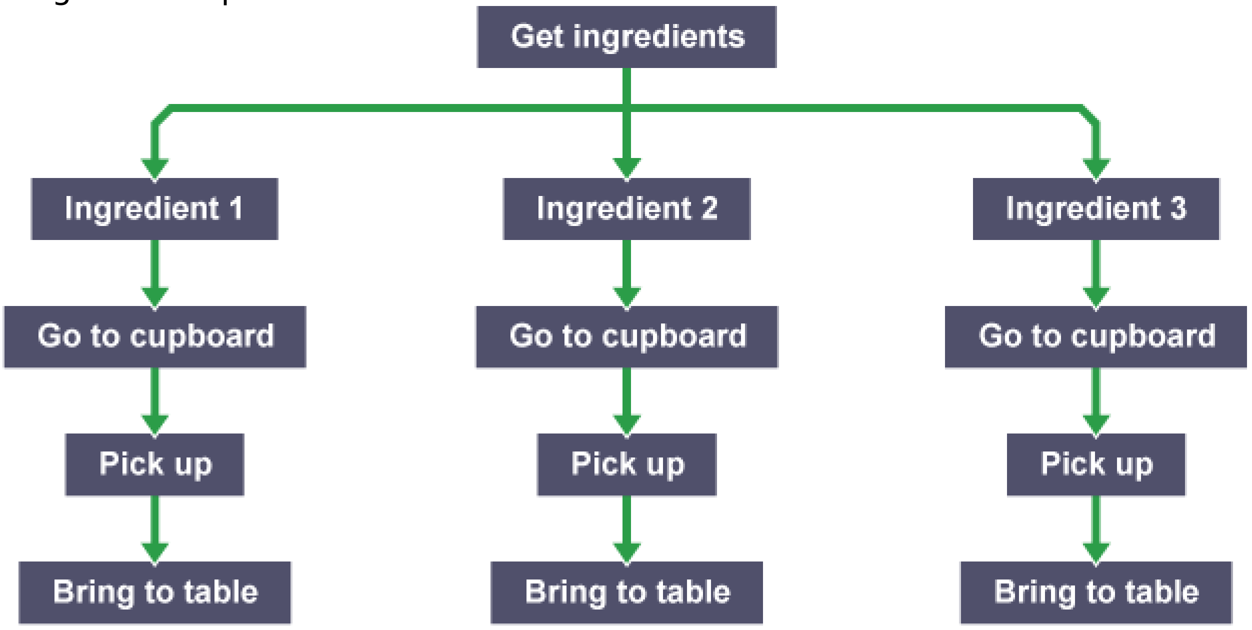
>

Solutions that are inefficient

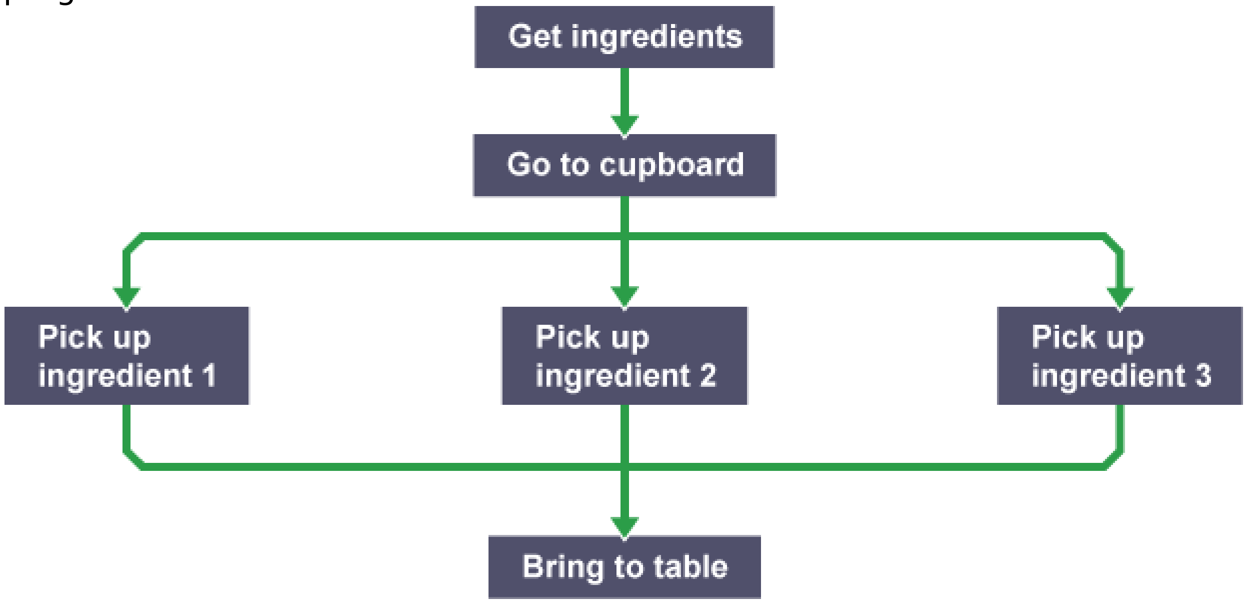
If **computational thinking** techniques are applied to the problem of how to bake a cake, on **decomposing** the problem, the solution would state – among other things – that certain quantities of particular ingredients are needed to make the cake.

For the first ingredient, it might tell us to go the cupboard, get the ingredient, and bring it back to the table. For the second – and all other ingredients – It might tell us to do the same.

If the cake had three ingredients, that would mean three trips to the cupboard. While the program would work like this, it would be unnecessarily long and complicated:



It would be more efficient to fetch all the ingredients in one go, and the program would be shorter as a result:



The solution is now simpler and more efficient, and has reduced from nine steps to five.

The problem occurred here because some steps were repeated unnecessarily, making the solution inefficient and overly long.

How do we evaluate our solution?

There are several ways to evaluate solutions. To be certain that the solution is correct, it is important to ask:

- does the solution make sense?

Do you now fully understand how to solve the problem? If you still don't clearly know how to do something to solve our problem, go back and make sure everything has been properly **decomposed**. Once you know how to do everything, then our problem is thoroughly decomposed.

- does the solution cover all parts of the problem?

For example, if drawing a cat, does the solution describe everything needed to draw a cat, not just eyes, a tail and fur? If not, go back and keeping adding steps to the solution until it is complete.

- does the solution ask for tasks to be repeated?

If so, is there a way to reduce repetition? Go back and remove unnecessary repetition until the solution is efficient.

Once you're happy with a solution, ask a friend to look through it. A fresh eye is often good for spotting errors.

Dry runs

One of the best ways to test a solution is to perform what's known as a 'dry run'. With pen and paper, work through the algorithm and trace a path through it.
For example, in **Algorithms**, a simple **algorithm** was created to ask someone their name and age, and to make a comment based on these. You could try out this algorithm – give it a dry run. Try two ages, 15 and 75. When using age 75, where does the algorithm go? Does it give the right output? If you use age 15, does it take you down a different path? Does it still give the correct output?

If the dry run doesn't give the right answer, there is something wrong that needs fixing. Recording the path through the algorithm will help show where the error occurs.

Dry runs are also used with completed programs. Programmers use dry runs to help find errors in their program code.